Chapter 10
Efficiency and Data Representation

Time granted does not necessarily coincide with time that can be most fully used.

— Tillie Olsen, *Silences*, 1965

Objectives

- To learn that different strategies for representing data can have a profound effect on the efficiency of your code.
- To be able to compare the computational complexity of different representation strategies.
- To understand the concept of a linked list and how it can be used to provide a basis for efficient insertion and deletion operations.
- To appreciate that representations which are efficient in terms of their execution time may be inefficient in their use of memory.

This chapter brings together two ideas that might at first seem to have little to do with each other: the notion of algorithmic efficiency presented in Chapter 8 and the idea of classes from Chapter 9. Up to now, efficiency has been closely linked with the study of algorithms. If you choose a more efficient algorithm, you can reduce the running time of a program substantially, particularly if the new algorithm is in a different complexity class. In some cases, choosing a different underlying representation for a class can have an equally dramatic effect. To illustrate this idea, this chapter looks at a specific class that can be represented in several different ways and contrasts the efficiency of those representations.

10.1 The concept of an editor buffer

Whenever you create a source file or use a word processor to edit text, you are using a piece of software called an **editor,** which allows you to make changes to a text file. Internally, an editor maintains a sequence of characters, which is usually called a **buffer.** The editor application allows you to make changes to the contents of the buffer, usually by some combination of the following operations:

- Moving the cursor to the point in the text at which editing needs to take place.
- Typing in new text, which is then inserted at the current cursor position.
- Deleting characters using a delete or backspace key.

Modern editors usually provide a highly sophisticated editing environment, complete with such fancy features as using a mouse to position the cursor or commands that search for a particular text string. Moreover, they tend to show the results of all editing operations precisely as they are performed. Editors that display the current contents of the buffer throughout the editing process are called **wysiwyg** (pronounced "wizzy-wig") editors, which is an acronym for "what you see is what you get." Such editors are very easy to use, but their advanced features sometimes make it harder to see how an editor works on the inside.

In the early days of computing, editors were much simpler. Lacking access to a mouse or a sophisticated graphics display, editors were designed to respond to commands entered on the keyboard. For example, with a typical keyboard-based editor, you insert new text by typing the command letter `I`, followed by a sequence of characters. Additional commands perform other editing functions, such as moving the cursor around in the buffer. By entering the right combinations of these commands, you can make any desired set of changes.

To make the idea of the editor buffer as concrete as possible, let's suppose that your task is to build an editor that can execute the six commands shown in Table 10-1. Except for the `I` command, which also takes the characters to be inserted, every editor command consists of a single letter read in on a line.

Table 10-1 Commands implemented by the editor

Command	Operation
`F`	Moves the editing cursor forward one character position
`B`	Moves the editing cursor backward one character position
`J`	Jumps to the beginning of the buffer (before the first character)
`E`	Moves the cursor to the end of the buffer (after the last character)
`I`*xxx*	Inserts the characters *xxx* at the current cursor position
`D`	Deletes the character just after the current cursor position

The following sample run illustrates the operation of the editor, along with annotations that describe each action. In this session, the user first inserts the characters **axc** and then corrects the contents of the buffer to **abc**. The editor program displays the state of the buffer after each command, marking the position of the cursor with a carat symbol (^) on the next line.

```
*Iaxc                    This command inserts the three characters 'a', 'x', and
 a x c                   'c', leaving the cursor at the end of the buffer.
     ^

*J                       This command moves the cursor to the beginning of the
 a x c                   buffer.
^

*F                       This command moves the cursor forward one character.
 a x c
   ^

*D                       This command deletes the character after the cursor.
 a c
   ^

*Ib                      This command inserts the character 'b'.
 a b c
     ^

*
```

10.2 Defining the buffer abstraction

In creating an editor that can execute the commands from Table 10-1, your main task is to design a data structure that maintains the state of the editor buffer. This data structure must keep track of what characters are in the buffer and where the cursor is. It must also be able to update the buffer contents whenever an editing operation is performed. In other words, what you want to do is define a new abstraction that represents the editor buffer.

Even at this early stage, you probably have some ideas about what internal data structures might be appropriate. Because the buffer is clearly an ordered sequence of characters, one seemingly obvious choice is to use a `string` or a `Vector<char>` as the underlying representation. As long as you have these classes available, either would be an appropriate choice. The goal of this chapter, however, is to understand how the choice of representation affects the efficiency of applications. That point is difficult to make if you use higher-level structures like `string` and `Vector` because the inner workings of those classes are not visible to clients. If you choose instead to limit your implementation to the built-in data structures, every operation becomes visible, and it is therefore much easier to determine the relative efficiency of various competing designs. That logic suggests using a character array as the underlying, because array operations have no hidden costs.

Although using an array to represent the buffer is certainly a reasonable approach to the problem, there are other representations that offer interesting possibilities. The fundamental lesson in this chapter—and indeed in much of this book—is that you should not be so quick to choose a particular representation. In the case of the editor buffer, arrays are only one of several options, each of which has certain advantages and disadvantages. After evaluating the tradeoffs, you might decide to use one strategy in a certain set of circumstances and a different strategy in another. At the same time, it is